Science of Computer Programming 7 (1986) 297-312
North-Holland

COMPILING AND OPTIMIZING METHODS FOR THE FUNCTIONAL LANGUAGE ASL/F

Katsuro INOUE

Department of Information and Computer Sciences, University of Hawaii, Honolulu, HI 96822, U.S.A.

Hiroyuki SEKI, Kenichi TANIGUCHI and Tadao KASAMI

Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University, Toyonaka, Osaka 560, Japan

Communicated by G. Berry
Received April 1984
Revised November 1985, January 1986

Abstract. A purely functional language called ASL/F is defined, and compiling and optimizing methods, by which ASL/F programs are translated into object programs that can be executed efficiently on the conventional machines, are studied. The effectiveness of those methods is investigated by implementing an optimizing compiler for ASL/F and executing several sample programs. Experimental results show that (1) all optimization techniques discussed here are useful in reducing the execution time and/or memory space requirement, and (2) the execution time of an ASL/F program is about 75 to 135% of that of a PASCAL program which implements the same algorithm.

1. Introduction

ASL/1 is an algebraic language we have designed, in which the semantics of a text is defined using congruence relation on expressions (terms) [16]. We have developed a verification support system for ASL/1 and verified the correctness of some algebraic specifications such as HDLC procedures using the support system [11]. Using ASL/1, we can write specifications in any abstract level, but in general there is no efficient method to compute 'the value' of a given expression.

As a sublanguage of ASL/1, we defined a purely functional programming language ASL/F by imposing restrictions on the form of the left-hand sides of axioms, and studied the methods to execute ASL/F programs efficiently. It is known that purely functional languages have good properties such as simply defined semantics and mathematical elegance [4, 8]. But the execution of functional language programs on a conventional machine has been considered to be less efficient in time than the execution of equivalent programs written in procedural languages. And the effects of optimizations for generating efficient object programs are scarcely reported.[1]

In this paper, it is shown that programs in ASL/F can be compiled into object programs that can be executed on a conventional machine as efficiently as object programs of equivalent PASCAL programs.

[1] As a related paper, [6] is pointed out by one of the referees.

1.1. Functional programming language ASL/F

Figure 1 shows a quicksort program written in ASL/F. A text of an ASL/F program consists of

(a) a program name (line (1) in Fig. 1),
(b) syntax declarations of defined functions (lines (3) to (10)),
(c) definitions of defined functions (lines (11) to (18)), and
(d) an expression to be evaluated (called the main program term, line (19)).

A syntax declaration of a defined function g has a form of $g: s_1, \ldots, s_n \rightarrow s$ which denotes that the number of arguments of g is n, the sort (data type listed in

```
(1) SPEC QUICKSORT(MAXLEN) ;
(2) INCLUDE ARRAY(INT, MAXLEN, ARY) ;
(3) OP QSORT        : ARY, INT, INT                    -> ARY ;
(4)    SPLIT&SORT   : ARY, INT, INT, INT, INT, INT -> ARY ;
(5)    LEFT         : ARY, INT, INT                    -> INT ;
(6)    RIGHT        : ARY, INT, INT                    -> INT ;
(7)    EXCH         : ARY, INT, INT                    -> ARY ;
(8)    MID          : INT, INT                         -> INT ;
(9)    INC          : INT                              -> INT ;
(10)   DEC          : INT                              -> INT ;
   AXIOM
(11)  QSORT(X, I, J) == IF( GE(I, J), X,
          SPLIT&SORT( X, I, J, I, J, CONTENT(X, MID(I, J))) )  ;
(12)  SPLIT&SORT(X, I, J, L, R, B) ==
        IF( LT(LEFT(X, L, B), RIGHT(X, R, B)),
           SPLIT&SORT( EXCH(X, LEFT(X, L, B), RIGHT(X, R, B)),
                       I,  J,
                       INC(LEFT(X, L, B)),  DEC(RIGHT(X, R, B)),
                       B  ),
           IF( EQ(LEFT(X, L, B), RIGHT(X, R, B)),
               QSORT( QSORT(X, I, DEC(RIGHT(X, R, B)) ),
                      INC(LEFT(X, L, B)),  J  ),
               QSORT( QSORT(X, I, RIGHT(X, R, B) ),
                      LEFT(X, L, B),  J  ) ) )                  ;
(13)  LEFT(X, L, B) ==
        IF( GE(CONTENT(X, L), B), L, LEFT(X, INC(L), B) )       ;
(14)  RIGHT(X, R, B) ==
          IF( LE(CONTENT(X, R), B), R, RIGHT(X, DEC(R), B) )    ;
(15)  EXCH(X, P1, P2) == ASSIGN(ASSIGN(X, P1,CONTENT(X, P2)),
                                P2, CONTENT(X, P1) )            ;
(16)  MID(I, J) == DIV(ADD(I, J), 2)                           ;
(17)  INC(I) == ADD(I, 1)                                      ;
(18)  DEC(I) == SUB(I, 1)                                      ;
   END                                                         ;
(19)  QSORT(X, 1, N)                                           ;

*  SPLIT&SORT(X, I, J, L, R, B)  sorts the elements X[I],
X[I+1], ..., X[J-1] and X[J] of array X under the assumption
that X[k] ≤ B for each k , I ≤ k ≤ L
and  X[l] ≥ B for each l , R ≤ l ≤ J .
```

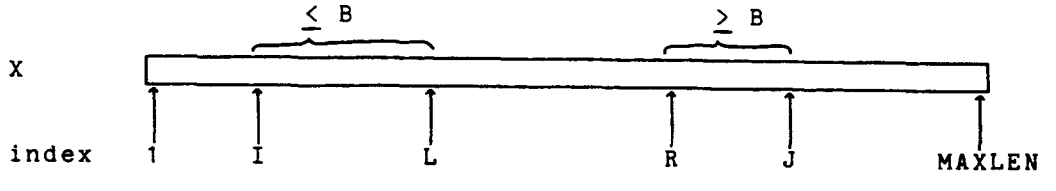

Fig. 1. Quicksort program written in ASL/F.

Table 1
Data types of ASL/F

Sort:	Boolean, Integer, Character, String, Real, Array, Tuple, File, Pointer, List, Structure, Union (of sorts)
Primitive functions:	Boolean: AND, OR, NOT, ... Integer: ADD, SUB, EQ, ... Array: CONTENT, ASSIGN, ...

In the compiler reported here, Integer, Boolean, Array of Integers, Tuples of them and 28 primitive functions are implemented.

Table 1) of the ith argument of g is s_i for $1 \leq i \leq n$ and the sort of the value of g is s. Each function definition has a form of $g(x_1, \ldots, x_n) \equiv right_g$ where g is a function name to be defined, $x_1, \ldots, x_n$ are distinct variables (formal parameters), and $right_g$ is a 'term' which consists of (1) primitive functions, (2) IF functions, (3) defined functions, (4) constants (TRUE, FALSE, ..., 0, 1, 2, ...), and (5) variables $x_1, \ldots, x_n$. Line (2) in Fig. 1 is a declaration of one dimensional array consisting of 1000 integers whose sort name is ARY and which has 'CONTENT' and 'ASSIGN' as the primitive functions (the value of CONTENT(X, i) is the ith element of array X, and ASSIGN(X, i, d) is the array obtained by replacing the ith element of X with d). Program text may have formal parameters, such as MAXLEN in line (1). For example, QUICKSORT(5000) denotes the text obtained from text QUICKSORT(MAXLEN) by replacing MAXLEN with 5000.

For an ASL/F program P, let $\approx_P$ denote the least congruence relation [10] on the set of all ground terms (terms without variables) generated by the following axioms:

(1) $\{p(c_1, \ldots, c_n) \equiv c \mid p$ is a primitive function, $c_1, \ldots, c_n$, c are constants and the value of $p(c_1, \ldots, c_n)$ is $c\}$;

(2) The axioms of IF function

$$\text{IF(TRUE}, x_1, x_2) \equiv x_1, \quad \text{and} \quad \text{IF(FALSE}, x_1, x_2) \equiv x_2$$

(3) All the definitions of defined functions in P regarded as axioms.

For any ground term t, if there exists a unique constant c satisfying $t \approx_P c$, then the value of t in P is defined as c, and undefined otherwise. The value of P for input data $d_1, \ldots, d_n$ is defined as the value of the term obtained from the main program term t_P in P by replacing all variables $x_1, \ldots, x_n$ appearing in t_P with $d_1, \ldots, d_n$ respectively.

We can regard axioms (1) to (3) as a set of rewrite rules, denoted R. Let $\Rightarrow_R^*$ denote the transitive-reflexive closure of the reduction relation [10] on the set of all ground terms associated with R. $\Rightarrow_R^*$ has Church-Rosser property [10, 18] since R is left linear and nonoverlapping [18]. For a ground term t and a constant c, the

value of t is c iff $t \Rightarrow_R^* c$. Furthermore, in ASL/F, we can get the value of a ground term t (if exists) by rewriting t in the outer-most manner (lazy evaluation) although, in general rewriting system, an optimal rewriting strategy to obtain the normal form [10] of a given term is not known. The operational aspect of the semantics of ASL/F is similar to that of the algorithmic specification described in [14].

Since the definition of semantics is simple and clear, verifying the correctness of a program and performing various kinds of optimizations are comparatively easy.

1.2. Survey of optimizations

In this paper, we adopt a compiling method such that a generated object program 'computes' the values of terms instead of rewriting terms directly, as will be described in Section 2.

The primary aim of the optimizations described here is to reduce the execution time and the dynamic memory requirement of the object program. Reducing such time and space is considered to be more important than reducing the static size of the object program and the time for compilation and optimization. We have formulated several optimization problems and implemented optimizers for these optimizations in the compiler [12, 13, 19]. These optimizations were chosen mainly by making comparison between PASCAL and non-optimized ASL/F object programs. Table 2 lists these optimizations and shows their effects which are obtained by executing some sample programs. Here, we explain the opimizations briefly, and the details will be described in Section 3.

(1) Pre-computation of arguments of defined functions: In order to obtain the value of an ASL/F program without fail, we adopt a method of 'lazy evaluation' [3, 7, 9] of arguments of defined functions as a basic strategy. In our implementation of lazy evaluation, for each argument t of a defined function, there is a subprocedure which computes and returns the actual value of the argument t, and the subprocedure is activated when the value of t turns out to be necessary for evaluating the function. To reduce the overhead of the activations of these subprocedures, we adopt the following method [12, 13, 19]. The ith argument of g is said to be *needed* if, in order to get the value of a term $g(t_1, \ldots, t_n)$, we must always compute the value of t_i regardless of the computation order and actual parameters $t_1, \ldots, t_n$. At compile-time, the compiler detects 'needed' arguments of a defined function, and generates an object code, where the values of needed arguments are precomputed and passed as actual parameters to a procedure for computing the function.

As seen in columns 1 and 2 of Table 2, this optimization is very effective in reducing the execution time and memory requirement (the maximum length of the run-time stack). In addition to these effects, detecting the needed arguments of defined functions will increase the possibilities of applying the following optimizations (2), (3), and (4).

(2) Avoidance of duplicate computations for common subterms: In order to avoid the duplicate computation for common subterms (subexpressions) within a term in

Table 2
List of optimizations and effects of them

Execution time (milli seconds)		(1)	(2)	(3)	(4)	(5)
		Optimization performed				
Program		a, b	a, b, c	a, b, c, d	a, b, c, d, e	a, b, c, d, e, f
Quicksort	5000 integers	3200	990	830	670	430
Bubblesort	50 integers	53	18	14	10	6
	1000 integers	[20 000]	7000	5900	3800	2400
Towers of Hanoi	10 stairs	160	20	20	15	10
	15 stairs	[5000]	630	630	460	350
Calculation of e	100 digits	250	64	52	48	23
	1000 digits	[14 000]	3500	3200	2800	1300
Multiplication of two (50, 50)-matrices		4700	1100	650	640	640

The maximum run-time stack length (word = 32 bits)		(1)	(2)	(3)	(4)	(5)
		Optimization method				
Program		a, b	a, b, c	a, b, c, d	a, b, c, d, e	a, b, c, d, e, f
Quicksort	5000 integers	73 700	49 900	529	433	433
Bubblesort	50 integers	31 600	225	27	24	17
	1000 integers	[$1.2 * 10^7$]	4030	27	24	17
Towers of Hanoi	10 stairs	37 800	194	194	163	162
	15 stairs	[$1.2 * 10^6$]	285	285	238	242
Calculation of e	100 digits	44 300	525	33	31	20
	1000 digits	[$2.8 * 10^6$]	3670	33	31	20
Multiplication of two (50, 50)-matrices		44 100	818	36	36	31

Optimization method:
a: Avoidance of duplicate computation for common subterms.
b: Globalization of arrays.
c: Pre-computation of needed arguments of defined functions.
d: Elimination of tail recursions.
e: Elimination of redundant flag tests.
f: Elimination of auxiliary functions.

Square brackets denote that the object programs could not be executed because of memory shortage. The values in them are our estimates.

a program, we use a flag indicating whether or not the value of the common subterms has been already computed. The flag is checked before the computation of the common subterms. The implemented compiler performs this optimization always. The optimized object program computes the value of each term according to rewriting of DAG's (Directed Acyclic Graphs) [3] instead of rewriting of terms (trees).

Some of these flag tests or flags themselves can be eliminated if they are known to be redundant by a control flow analysis at compile-time. The effect of this elimination of redundant flag tests is shown in columns 3 and 4 of Table 2.

(3) Globalization of sorts: For a sort (data type) such as array which requires a large memory space to store its value, no memory is allocated for the values of the sort dynamically and a fixed memory space sufficient to store any single value of the sort is allocated statically if the source program satisfies a certain condition (which guarantees that such an object program can compute the value of the program correctly, see Section 3.3). This optimization (called globalization of sorts) is essential to purely functional languages from a practical point of view. If this is not performed, arrays will be copied repeatedly on the run-time stack in general, and memory space and execution time would be exhausted. Our compiler generates an object program only if sorts of arrays in a source program are globalizable.

(4) Elimination of tail recursions: Defined function activations having tail recursive forms are transformed into iterative forms as seen in functional and procedural languages [2, 5]. The compiler detects those forms by analyzing the occurrences of defined function activations and the computation order of their arguments (e.g., whether they are needed or not) [13]. This optimization greatly reduces the memory requirement as seen in columns 2 and 3 in Table 2.

(5) Elimination of auxiliary functions: If in the source text of an ASL/F program, there are subterms which can be rewritten without knowing the values assigned to variables, we can then rewrite (expand) such subterms repeatedly before compilation and other optimizations. This optimization corresponds to the expansion of open subroutines in procedural languages, and reduces the number of procedure activations. And also it may increase the number of common subterms; hence, the duplicate computation can be avoided by optimization (2). The effect of this optimization is seen in columns 4 and 5 of Table 2. This optimization reduces the execution time, but it might sometimes increase the dynamic memory requirement because it increases the size of the right-hand side terms of definitions.

For these optimizations, only the source text of a program is analyzed. Many ordinary optimization techniques which are applied to procedural languages such as FORTRAN (e.g., register allocation, loop-invariant move, etc.) can be used to improve further the object program generated by our compiling and optimizing methods. But these optimizations are not considered here.

In the optimizing compiler reported here, only boolean, integer, array of integers and their primitive functions were implemented because our aim was to investigate the effectiveness of our optimizations. It runs under the UTS/VS operating system on a MELCOM COSMO 900-II (it executes about 6 million instructions per second),

and it compiles an ASL/F program into an object program in an assembly language META-SYMBOL. The compiler was written in PASCAL, and is about 4000 lines long (including parsing, optimizing, and code generation routines). It took about 7 man-months to design and implement this compiler. An ASL/F compiler which implements all the data types listed in Table 1 is under design.

2. Implementation

2.1. Evaluation order of terms

For a given term $t = f(t_1, t_2, \ldots, t_n)$, in order to obtain the value of t without fail when t has a value, each subterm is evaluated in the following order:

(1) If f is a primitive function, all of $t_1, t_2, \ldots, t_n$ are evaluated before f itself is evaluated. There is no constraint of the evaluation order among $t_1, t_2, \ldots, t_n$ to obtain the value of t; however, it may be determined to satisfy optimization conditions such as those for the globalization of sorts.

(2) If f is an IF function, that is, $t = \mathrm{IF}(t_1, t_2, t_3)$, then t_1 is at first evaluated. Depending on the result of evaluating t_1, either t_2 or t_3 is evaluated.

(3) If f is a defined function, f itself is evaluated first. Each t_i is evaluated only when its value turns out to be necessary for evaluating f at execution time. The order will be modified in the optimization described in Section 3.1.

2.2. Object programs

In this section, we describe an object program which computes, for given input values, the value of the main program term, in the computation order described above.

Here, for the main program term t_P, we introduce an additional definition $\mathrm{MAIN}(x_1, \ldots, x_n) \equiv t_P$ (where $x_1, \ldots, x_n$ are distinct variables in t_P) in order to simplify the following discussions.

For an ASL/F program P with main program term t_P with n distinct variables, the compiler generates an object program (written in a procedural langauge such as an assembly language, for example) which computes the value of t_P with input data $d_1, \ldots, d_n$ assigned to the variables. The object program consists of

(1) master procedure F_g for each defined function g, which computes and returns the value of function g,

(2) slave procedure H_t for each argument t of a defined function, which computes and returns the actual value of argument t, and

(3) the main program corresponding to the main program term t_P.

The value computed and returned in each procedure is obtained by executing the following instruction sequence. Here we assume that $t = f(t_1, \ldots, t_n)$ is a term whose value is to be computed and returned in the procedure.

CODE(t) ≡
  case f of
    'primitive function p': CODE(t_1), ..., CODE(t_n), and code to compute p with argument values, where each of CODE(t_1), ..., CODE(t_n) is evaluated in a certain order before computing p;
    'constant c' : code to generate c;
    'IF function' : CODE(t_1), CODE(t_2), CODE(t_3), where CODE(t_2) is executed if the execution result of CODE(t_1) is TRUE, and CODE(t_3) is executed otherwise;
    'defined function g' : code to activate a master procedure F_g (CODE(t_1), ..., CODE(t_n) are contained in different slave procedures, $H_{t_1}, \ldots, H_{t_n}$.) Parameters to the master procedure are entry addresses of $H_{t_1}, \ldots, H_{t_n}$;
    'variable x' : CODE to activate a slave procedure whose entry address has been passed as a parameter.

The computation of the value for the main program term is proceeded by the activations of master procedures and slave procedures. When the value of a term such as $g(\ldots, t, \ldots)$ (where g is a defined function) is computed, slave procedure H_t is activated from master procedure F_g only when the value of t turns out to be necessary for evaluating $g(\ldots, t, \ldots)$ at execution time.

The slave procedures are similar to 'implicit subroutines' which evaluate arguments passed by name in the object program of an ALGOL program [17], but the slave procedures work more simply.

The environments and working space (called a *frame*) are allocated on a single ordinary LIFO stack when each procedure is activated. It is not necessary to implement a 'display' technique or environment list as used in ALGOL or LISP [1], since ASL/F does not have scope rules or binding rules.

3. The details of optimizations

3.1. Pre-computation of needed arguments of defined functions

If an argument of a defined function g is detected to be needed, the object program is modified so that the value of the argument is computed before the activation of master procedure F_g for g. The computed value is passed to F_g as an actual parameter (passed by value).

Here, we give a sufficient condition for an argument of a defined function to be needed. For simplicity, let the definition of each defined function g in a given ASL/F program be $g(x_1, \ldots, x_{n_g}) \equiv right_g$. For each defined function g, each subset I of set $I_g = \{1, 2, \ldots, n_g\}$ and each subterm t of $right_g$, we introduce a boolean variable $Y[t, I]$ which is used to denote whether or not for any substitution of terms for variables in t, there exists an integer i in I (possibly depending on the substitution) such that we must know the value of x_i in order to get the value of t (if there exists such an integer i in I, we say that I is needed for t)

We set up a system of equations for $Y[t, I]$'s defined as follows:

(a) If t is a constant, $Y[t, I] = \text{FALSE}$;

(b) If t is a variable in I, $Y[t, I] = \text{TRUE}$;

(c) If t is a variable not in I, $Y[t, I] = \text{FALSE}$;

(d) If t is $\text{IF}(t_1, t_2, t_3)$,

$$Y[t, I] = Y[t_1, I] \vee \{Y[t_2, I] \wedge Y[t_3, I]\};$$

(e) If t is $f(t_1, \ldots, t_{n_f})$ where f is a primitive function,

$$Y[t, I] = Y[t_1, I] \vee \cdots \vee Y[t_{n_f}, I];$$

(f) It t is $f(t_1, \ldots, t_{n_f})$ where f is a defined function,

$$Y[t, I] = \bigvee_{\substack{\text{for each subset } J \\ \text{of } \{1,\ldots,n_f\}}} \{Y[\text{right}_f, J] \wedge \bigwedge_{\substack{\text{for each} \\ k \text{ in } J}} Y[t_k, I]\}.$$

For example, definition (d) means that I is needed for t if it is needed in the first argument (predicate) or both in the second (TRUE case) and the third (FALSE case) and definition (f) means that I is needed for t if there exists an integer set J, such that (1) J is needed for $right_f$ and (2) I is needed for each t_k with k in J.

Let E_P be the system of equations for Y defined above. If there is a solution of E_P such that $Y[right_g, \{i\}] = \text{TRUE}$, then the ith argument of g is a needed argument. The proof is given in [19].

Let $>$ be the order on {TRUE, FALSE} such that TRUE $>$ FALSE. Since there are no complementations in the right-hand sides of the equations in E_P, E_P has always the maximum solution with respect to $>$.

An $O(n)$-time algorithm for finding the maximum solution of E_P, where n is the number of subterms appearing in the right-hand sides of definitions in a given ASL/F program,[2] is as follows. For each $Y[t, I]$ we introduce a program variable which will be also denoted $Y[t, I]$.

(1) For each $Y[t, I]$, set $Y[t, I]$ to FALSE if t is a constant or a variable x_i with i not in I (see (a) and (c) in the definition of E_P above) and TRUE otherwise.

(2) Repeat (*) until no $Y[t, I]$ changes to FALSE.

[2] Here, we have a natural assumption that the maximum number of arguments of defined functions does not increase depending on n.

Apply (**) to all pairs of t of the form $f(t_1, \ldots, t_{n_f})$ and I such that (i) $Y[t, I]$ = TRUE and (ii) $Y[t_i, I]$ for some t_i or $Y[right_f, J]$ for some subset J of $\{1, \ldots, n_f\}$ was set to FALSE in the last execution of (*). (*)

(For the first execution of (*), "the last execution of (*)" means the execution of (1).)

Test whether the right-hand side of the equation in E_P whose left-hand side is boolean variable $Y[t, I]$ (see (d) to (f) in the definition of E_P above) is TRUE or FALSE, and if it is FALSE, then set program variable $Y[t, I]$ to FALSE. (**)

(3) Output all pairs $\langle g, k \rangle$ (k is an integer in I_g) such that $Y[right_g, \{k\}]$ = TRUE.

If a pair $\langle g, k \rangle$ is in the output of this algorithm, the kth argument of defined function g is needed. A detailed description of this algorithm and analysis of its complexity are given in [19].

An idea analogous to ours was presented in earlier paper by Mycroft [15] and the sufficient condition in [15] is logically equivalent to ours. Mycroft's algorithm involves symbolic manipulations of boolean expressions and decision procedures whether given two boolean expressions represent the same function, and no analysis of the complexity was given in [15].

In our compiler, the optimizer detects needed arguments by finding the maximum solution of $\bar{E}_P$ where $\bar{E}_P$ is the same as E_P except that $Y[t, I]$'s are defined only for I which is a singleton and we restrict subset J to singleton in the 'or' operation in the right-hand side of (f) of the definition of E_P. Although this sufficient condition using $\bar{E}_P$ is weaker than that of E_P, our optimizer found all of the needed arguments in sample programs listed in Section 4.1.

3.2. Avoidance of duplicate computation for common subterms

In order to avoid duplicate computation of (identical) values of common subterms, we modify the instruction sequence in each procedure as follows:

For a defined function f and a right-hand side term of the definition of f, we group together all the subterms of $right_f$ which are identical, and for each group (we call such a group a set of common subterms hereafter) C, extra fields VALUE_C and FLAG_C are allocated in each frame, where

- VALUE_C is a field for storing a value of the subterms, and
- FLAG_C is a field for storing a flag indicating whether the value to be stored in VALUE_C is "already computed" or "not yet computed". Initially FLAG_C is set to "not yet computed".

Before executing an instruction sequence CODE(t) for a subterm t in C, the flag in FLAG_C is checked. If the flag is "not yet computed", then CODE(t) is executed, and after the execution of CODE(t), the value computed is stored in VALUE_C, and FLAG_C is set to "already computed". Otherwise CODE(t) is not executed but VALUE_C is only referred to.

Let CMN-CODE(t) denote the modified instruction sequence of CODE(t) as described above. In CMN-CODE(t) redundant flag tests can be deleted as follows. Let C be a set of common subterms $\{t_1, t_2, \ldots, t_m\}$ which have the same value. If the result of the flag test in instruction sequence CMN-CODE(t_i) is always "not yet computed" for any input values of a program, then t_i is called "always the first", and the instruction for the flag test is deleted from CMN-CODE(t_i), so that the computation of the value of t_i can start immediately. Similarly, if the result of the flag test in instruction sequence CMN-CODE(t_j) is always "already computed" for any input values of a program, then t_j is called "always the second or later". In such a case, an instruction for referring to the value already computed is only necessary. If each common term in C is either "always the first" or "always the second or later", then the field for the flag is not necessary.

In order to find subterms t_i's which are "always the first" or "always the second or later", the compiler analyzes the control flow of instruction sequence CMN-CODE(r) where r is the smallest term including occurrences of all $t_1, t_2, \ldots, t_m$. A sufficient condition for the elimination of redundant flag tests is described in details in [19].

3.3. Globalization of sorts

For a set of sorts S, we say that S is *globalizable* if there exists a set of instruction sequences $\{I_f |$ for every defined function $f\}$ where I_f computes the value of f in such a way that for every sort s in S, a current value of s is always stored in a fixed space assigned for the sort s. For a sort s in a globalizable set of sorts, we modify the object program as follows:

(1) At compile-time, we statically allocate space W_s sufficient to store any single value of sort s;

(2) For each instruction whose execution generates a value of sort s (including an instruction sequence to read input value of sort s), we store the result value into W_s, and for each instruction sequence which refers to a value of sort s, we read the value of sort s from W_s.

In principle, this optimization is related to the "register allocation" problem applied to procedural languages [1] and in a particular case where every argument of every defined function is specified as "pre-computation", it corresponds to "one pebbling problem" [20].

Our sufficient condition for a set S of sorts to be globalizable is that the following hold for each s in S:

(1) All functions except for IF function have at most one argument of sort s;

(2) For a defined function g, let GEN(g, s) denote the set of nodes v's in the DAG representing term $right_g$ such that some value of sort s may be generated in computing the value of (the term represented by) v. Then the following hold:

(2.1) In GEN(g, s) there is at most one node without a child of sort s;

(2.2) If distinct nodes u and w in GEN(g, s) have a common child of sort s, then at most one of the computations for u and w is executed for any input data;

(2.3) For each node v of sort s, all the references to the value of v have been made before a value of sort s is generated in computing the value of a parent of v.

The details are described in [19].

4. Analysis

4.1. *Effects of optimizations*

We investigated the effectiveness of the optimizations by executing several sample programs. Algorithms for solving the following problems have been programmed in ASL/F:

(1) Sorting (Quicksort): the program is shown in Fig. 1,
(2) Sorting (Bubblesort),
(3) Towers of Hanoi,
(4) The computation of the base of natural logarithm, 'e',
(5) Matrix multiplication.

The programs are written in such a way that they may be considered to be 'natural implementations' of the algorithms in ASL/F.

Table 2 shows the execution time and the maximum run-time stack length of those ASL/F programs. These experimental results demonstrate that all of the optimizations adopted here are useful in reducing the execution time and/or the maximum run-time stack length. Especially, the following are remarked. (In the following (i), optimizations "avoidance of duplicate computation for common subterms" and "globalization of arrays" are always performed and in (ii) "pre-computation of needed arguments" besides these two optimizations are always performed.)

(i) Pre-computation of needed arguments of defined functions is effective to reduce both the execution time and the maximum run-time stack length greatly. For example, if the pre-computation of needed arguments is not performed in the program to solve Towers of Hanoi, the maximum run-time stack length is an order of exponential of n, where n is the number of stairs. On the other hand, if it is performed, the maximum run-time stack length is linear in n as shown in Table 2.

(ii) If the elimination of tail recursions is possible, then it reduces the maximum run-time stack length considerably. For example, if the elimination of tail recursions is not performed in Bubblesort program, then the maximum run-time stack length is approximately linear in the number of integers to sort. On the other hand, if it is performed, the maximum run-time stack length is fixed to a constant, 27 words.

4.2. *Comparison between ASL/F and PASCAL programs*

ASL/F programs mentioned above, and PASCAL programs, which implement the same algorithms as ASL/F programs were executed on the same machine. The execution time is shown in Table 3. These PASCAL programs are also natural

Table 3
Execution time of ASL/F and PASCAL programs (milliseconds)

Program (data size)	ASL/F	PASCAL
Quicksort (5000 integers)	430[a]	320[b]
Bubblesort (50 integers)	6	8
(1000 integers)	2400	3200
Towers of Hanoi (10 stairs)	10	11
(15 stairs)	350	370
The calculation of the base of natural logarithm (100 digits)	23	28
(1000 digits)	1300	1500
Multiplication of two matrices (50 * 50)	640	820

[a] This program is shown in Fig. 1.
[b] This program is by Wirth [22] and shown in Fig. 2.

implementations of the algorithms, where iterations such as FOR, WHILE, etc. are used and some duplicate computations are eliminated by using variables to store temporary values. For example, the quicksort program in PASCAL used here is presented by Wirth [22] and shown in Fig. 2. When those ASL/F programs were compiled, all the optimizations described in Section 3 were performed. PASCAL programs were compiled by a MELCOM PASCAL 8000(A20) compiler which generates machine codes directly. Table 3 shows that the execution time of an ASL/F program can be considered to be almost the same as that of the corresponding PASCAL program. The time required for compiling the quicksort program, for

```
PROCEDURE QUICKSORT ;
VAR  N    : INTEGER ;

PROCEDURE SORT(L, R :INTEGER) ;
VAR I, J, M, W :INTEGER ;
BEGIN
  I:=L ;  J:=R ;  M:=X[(L+R) DIV 2 ] ;
  REPEAT
     WHILE X[I] < M DO I:=I+1 ;
     WHILE M < X[J] DO J:=J-1 ;
     IF I<=J THEN BEGIN
       W:=X[I] ; X[I]:=X[J] ; X[J]:=W ;
       I:=I+1 ; J:=J-1
     END
  UNTIL I>J ;
  IF L < J THEN SORT(L, J) ;
  IF I < R THEN SORT(I, R)
END ;

BEGIN  SORT(1, N)  END
```

Fig. 2. Quicksort program written in PASCAL.

example, was as follows:

(1) From ASL/F to the object program in the assembly language: 0.6 second (including 0.1 second for optimizations);

(2) From PASCAL to machine codes: 0.3 second.

4.3. Writing an interpreter in ASL/F

As an example of fairly long ASL/F programs, we wrote an ASL/F program which interprets (parses and executes interpretively) an ASL/F program. Table 4 shows the size characteristics of this interpreter program. It took about 2 months for a senior student to develop this interpreter program. At the beginning of its development, he had several syntactic errors but only two logical errors. The execution time of this interpreter for Ackermann function ACK(3, 3), for example, was 3.6 seconds. The program of the Ackermann function was compiled by the optimizing compiler and executed also. It took 0.01 second to compute ACK(3, 3).

Table 4
Size characteristics of the ASL/F interpreter written in ASL/F

		Parsing part	Execution part
The number of defined functions		110	70
The number of arguments of defined functions	maximum	10	6
	average	4	4
The nesting depth of functions in the right-hand sides of definition statements	maximum	7	11
	average	3	3
The number of lines in the text		350	200

5. Conclusion

We have discussed the compiling and optimizing method for functional programming language ASL/F and shown that those methods are effective to increase the time and space efficiency. The execution time of an ASL/F program is about 75 to 135% of that of a PASCAL program which implements the same algorithm.

If conventional optimization techniques adopted in compilers of procedural languages are applied to the object program generated by our compiler, the time and space efficiency will be increased further.

The compiler reported here adopts the right-to-left evaluation order among the arguments of a function. However, the execution time and memory space requirement will be reduced further if the compiler could find a better evaluation order for each function, in the sense that more sorts in the source program satisfy the sufficient condition to be globalizable and that more common terms satisfy the sufficient condition to be "always the first" or "always the second or later".

It is an important research topic to investigate a method of transforming a specification written in ASL/1 into an efficient ASL/F program. A specification, written in the style of "Abstract Sequential Machine", can be transformed straightforwardly into an ASL/F program [21].

Acknowledgment

We gratefully acknowledge many helpful suggestions provided by Dr. Yuji Sugiyama of Department of Information and Computer Sciences, Osaka University.

References

[1] A.V. Aho and J.D. Ullman, *Principles of Compiler Design* (Addison-Wesley, Reading, MA, 1977).
[2] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data Structures and Algorithms* (Addison-Wesley, Reading, MA, 1983) 66-67 and 78-82.
[3] G. Berry and J.J. Levy, Minimal and optimal computation of recursive programs, *J. ACM* **26** (1) (1979) 148-175.
[4] G. Berry, Programming with concrete data structures and sequential algorithms, *Proc. ACM Conference on Functional Programming Languages and Computer Architecture* (1981).
[5] R.M. Burstall and J. Darlington, A translation system for developing recursive programs, *J. ACM* **24** (1) (1977) 46-67.
[6] G. Cousineau, P.-L. Curien and M. Mauny, The categorical abstract machine, *Proc. International Symposium on Functional Programming Languages and Computer Architecture*, Nancy, Lecture Notes in Computer Science **201** (Springer, Berlin, 1985) 50-64.
[7] D.P. Friedman and D.S. Wise, CONS should not evaluate its arguments, *Proc. 3rd ICALP*, Edinburgh (1976) 257-281.
[8] P. Henderson, *Functional Programming, Application and Implementation* (Prentice-Hall, Englewood Cliffs, NJ, 1980) 218-223.
[9] P. Henderson and J.H. Morris, A lazy evaluator, *Proc. 3rd POPL* (1976) 95-103.
[10] G. Huet and D.C. Oppen, Equations and rewrite rules: A survey, in: R. Book, Ed., *Formal Languages: Perspectives and Open Problems* (Academic Press, New York, 1980) 349-393.
[11] T. Higashino, M. Mori, Y. Sugiyama, K. Taniguchi and T. Kasami, An algebraic specification of HDLC procedures and its verification, *IEEE Trans. Software Engrg.* **10** (6) (1984) 825-836.
[12] K. Inoue, H. Seki, Y. Sugiyama and T. Kasami, Code optimization at compilation of functional programming language ASL-F programs, *Papers of Technical Group on Electronic Computers*, IECE Japan EC82-18 (1982) 61-72 (in Japanese).
[13] K. Inoue,H. Seki, K. Taniguchi and T. Kasami, Functional programming language ASL/F and its optimizing compiler, *Trans. IECE Japan* **67-D** (4) (1984) 458-465 (in Japanese).
[14] J. Loecks, Algorithmic specifications of abstract data types, *Proc. 8th ICALP*, Lecture Notes in Computer Science **115** (Springer, Berlin, 1981) 129-147.
[15] A. Mycroft, The theory and practice of transforming call-by-name into call-by-value, *International Symposium on Programming*, Lecture Notes in Computer Science **83** (Springer, Berlin, 1980) 269-281.
[16] M. Nakanishi, Y. Sugiyama, K. Taniguchi, T. Kasami and W.W. Peterson, A system supporting program design—program derivation from an algebraic specification, *1979 National Convention Record on Information and System Section*, IECE Japan (1979) 402 (in Japanese).
[17] B. Randell and L. Russell, *ALGOL 60 Implementation* (Academic Press, London 1964) 98-106 and 214.
[18] B.K. Rosen, Tree manipulating systems and Church-Rosser theorems, *J. ACM* **20** (1) (1973) 160-187.

[19] H. Seki, K. Inoue, K. Taniguchi and T. Kasami, Optimization of functional language programs—optimizing compiler for ASL/F, *Trans. IECE Japan* **67-D** (10) (1984) 1115-1122 (in Japanese).

[20] R. Sethi, Pebble game for studying storage sharing, *Theoret. Comput. Sci.* **19** (1982) 69-84.

[21] K. Torii, Y. Morisawa, Y. Sugiyama and T. Kasami, Functional programming and logical programming for the telegram analysis problem, *Proc. 7th International Conference on Software Engineering* (1984) 463-472.

[22] N. Wirth, *Algorithms + Data Structures = Programs* (Prentice-Hall, Englewood Cliffs, NJ, 1976) 76-82.